Omówienie zadań 22 turnieju drużynowego

Autorzy: Olaf Surgut, Krzysztof Olejnik

5 października 2024

1 Balans smaku

Znanym faktem jest, że $2^n > 2^n - 1 = 2^0 + 2^1 + \ldots + 2^{n-1}$, co oznacza tyle że waga ostatniego kroku przeważa nad wartością reszty. W tym zadaniu wystarczało sprawdzić co dosypujemy w ostatnim kroku. Podczas zawodów wszelkie próby sumowania 2^i w C++ spotykało się z werdyktem *Błędna odpowiedź*, ze względu na przekraczanie maksymalnej wartości *int* (2^{31}), *long long* (2^{63}), czy nawet *unsigned long long* (2^{64}).

```python
n = int(input())
s = input()

if s[-1] == 'S':
    print("SLONA")
else:
    print("SLODKA")
```

2 Literkowa zupa

Zadanie można rozwiązać na co najmniej 2 sposoby:

1. Zastosujemy algorytm przeszukiwania z nawrotami, poniższa funkcja rekurencyjna sprawdza każdą możliwą ścieżkę w każdym kroku upewniając się, że wchodzimy na nieużyte jeszcze pole z literą, której aktualnie potrzebujemy.

```cpp
string slowo;
char zupa[3][3];
const int dy[] = {0, 1, 0, -1};
const int dx[] = {1, 0, -1, 0};
bool vis[3][3];

void dfs(int y, int x, int d){
    if(slowo[d] != zupa[y][x]){
        return;
    }
    vis[y][x] = 1;
    if(d == 8){
        cout << "TAK\n";
        exit(0);
    }
    for(int k=0;k<4;k++){
        int ny = y + dy[k];
        int nx = x + dx[k];
        if(ny<0||ny>=3||nx<0||nx>=3||vis[ny][nx]) continue;
        dfs(ny,nx,d+1);
    }
    vis[y][x] = 0;
}
```

2. Można dokonać następującą obserwację: "Jest tylko 8 ścieżek zaczynających się w (1,1) oraz przechodzących przez wszystkie pola.". Obserwacja ta pozwala nam zaimplementować rozwiązanie, które

polega na wpisaniu bezpośrednio do kodu wszystkich ścieżek i sprawdzenie czy którakolwiek z nich jest tą poprawną.

3 Krótkie menu

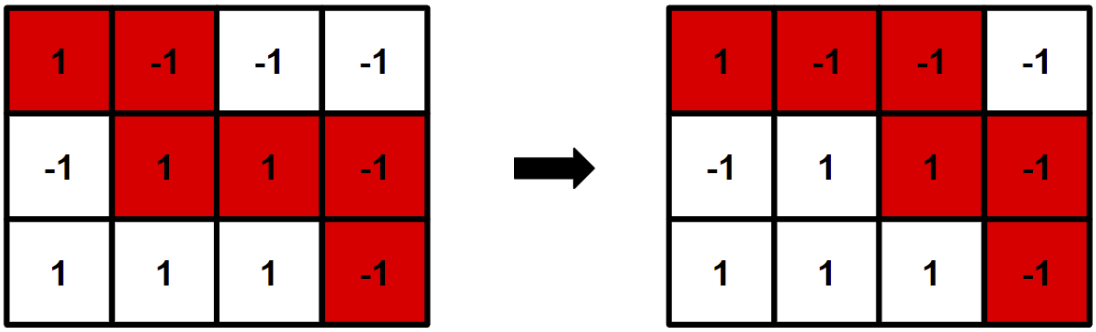
Przeanalizujmy, jak wygląda szkolny "algorytm" sumowania A oraz B i sprawdźmy jaki to ma wpływ na wynik. Ustawiamy A i B pod sobą, a następnie skanujemy od prawej do lewej i dodajemy cyfry w kolumnie. Jeśli suma ta przekroczy 9 to odejmujemy od aktualnej kolumny 10 i dodajemy 1 do następnej. Co to oznacza dla sumy cyfr $s(A)$ oraz $s(B)$? Za każdym przekroczeniem 9 w sumowaniu suma cyfr w $A+B$ spada o 9, a więc: $s(A+B) = s(A) + s(B) - 9 * (\textit{ile razy przekroczyliśmy } 9)$. Czyli $s(A) + s(B) = s(A+B) + 9 * x$. Zadanie sprowadza się do konstrukcji A oraz B z największą ilością przekroczeń 9 podczas dodawania. Jedna z takich konstrukcji jest postaci $A = 999\ldots999$, a $B = N - A$. Wystarczy teraz przejść po długości A i brać maksimum do odpowiedzi.

```
def s(X):
    if X == 0:
        return 0
    return s(X // 10) + (X % 10)

N = int(input())
ans = s(0) + s(N)

i = 9
while i <= N:
    ans = max(ans, s(i) + s(N - i))
    i = i * 10 + 9

print(ans)
```

4 Dieta cud

Zamieńmy 'P' na $+1$ oraz 'S' na -1. Teraz należy znaleźć ścieżkę z $(1,1)$ do (N,M) o sumie zerowej.

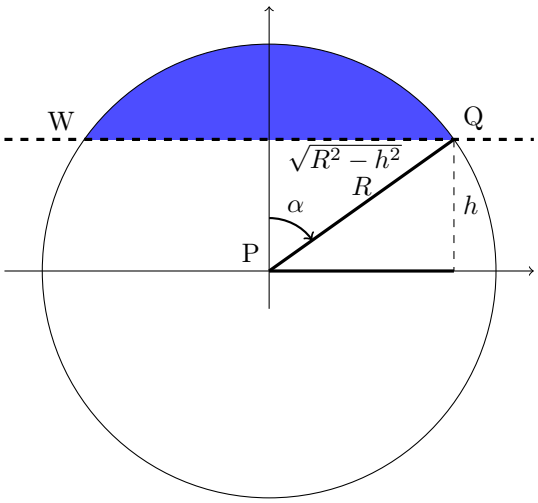
Warunkiem koniecznym i wystarczającym, aby odpowiedź istniała jest, aby $(N+M-1) \mod 2 = 0$ oraz ścieżka o min koszcie $\leq 0 \leq$ ścieżka o max koszcie.



Powiedzmy, że trzymamy w ręce min oraz max ścieżkę. Każda z nich jest stworzona z $N-1$ ruchów w dół oraz $M-1$ ruchów w prawo. Możemy przestawić sąsiednie ruchy, aby otrzymać jedno w drugie. Podczas każdej zamiany, suma na ścieżce zmieni się o -2, 0 lub $+2$. Więc po wykonywaniu operacji po kolei będzie moment, w którym ścieżka będzie miała sumę 0.
Wyznaczenie min i max ścieżki można za pomocą dwóch *dp*-ków, $MIN[x][y] = \min$ ścieżka od $(1,1)$ *do* (x,y), czyli $MIN[x][y] = \min\{MIN[x-1][y], MIN[x][y-1]\}$. Symetrycznie dla maksimum. Ścieżkę można odzyskać cofając się od (N,M) i sprawdzając czy w lewo lub w do góry jest spełniony warunek istnienia ścieżki o danej sumie.

Złożoność $O(NM)$

5 Kto tak kroi pizzę?



Aby rozwiązać zadanie należało najpierw wyznaczyć funkcję, która będzie obliczała powyższy obszar zamalowany na niebiesko. W tym celu można wyliczyć wycinek koła o kącie $2\alpha = 2\arccos(\frac{h}{R})$ oraz odjąć od tego trójkąt PQW o podstawie $2 \times \sqrt{R^2 - h^2}$ i wysokości h. Co finalnie daje

$$P(h) = \frac{2\alpha}{2\pi} \times \pi R^2 - 2 \times \frac{h\sqrt{R^2-h^2}}{2} = \alpha R^2 - h\sqrt{R^2-h^2} = \arccos(\frac{h}{R})R^2 - h\sqrt{R^2-h^2}$$

Jeśli $h < 0$, wystarczy od pola pizzy odjąć wynik dla $-h$.

Posiadając już taką funkcję w ręku można posłużyć się wyszukiwaniem binarnym i znajdować kolejne wysokości na których jest przynajmniej $\frac{i+1}{N}$ całkowitego pola. Po 40 iteracjach odpowiedź będzie odległa od optymalnej o maksymalnie $R \times 2^{-40} < 10^{-8}$. Złożoność $O(40N)$

```python
from math import acos, sqrt, pi

n, R = map(int, input().split())

def pole(h):
    if h < 0:
        return pi * R * R - pole(-h)
    return acos(h / R) * R * R - h * sqrt(R * R - h * h)

for i in range(n - 1):
    l, r = -R, +R
    need = pi * R * R * (i + 1) / n
    for iter in range(40):
        m = l + (r - l) / 2

        if pole(m) > need:
            l = m
        else:
            r = m

    h = l + (r - l) / 2
    print(f"{h:.8f}")
```

6 Wisienka na torcie

Prostym do pokazania jest fakt, że nie jesteśmy w stanie zużyć całego lukru i jednocześnie pokryć nim całego ciasta, jeżeli pole ciasta bez wisienki nie jest równe polu, które jesteśmy teoretycznie w stanie

pokryć lukrem.

Okazuje się, że gdy te pola są równe to rozwiązanie istnieje i realizuje je następujący rekurencyjny algorytm zachłanny: Jeżeli największym dostępnym lukrem można pokryć idealnie cały aktualny kawałek ciasta (bez pokrycia wisienki) to to zrób. W przeciwnym wypadku podziel ciasto na 4 kwadratowe ćwiartki, wywołaj się we wszystkich kawałkach bez wisienki, a następnie w ćwiartce z wisienką (o ile takowa istnieje).

Dowód działania jest stosunkowo prosty. Załóżmy że pole ciasta = pole, które można pokryć dostępnym lukrem. Zauważmy, że gdy wywołamy się w ćwiartce z wisienką to tak naprawdę rozwiązujemy problem dla N mniejszego o 1. Czyli tak naprawdę wystarczy pokazać, że wywołania w ćwiartkach bez wisienki przebiegną pomyślnie, aby pokazać poprawność całego algorytmu. Załóżmy, że coś poszło nie tak. Jedyną sytuacją, w której algorytm by się nie zakończył powodzeniem jest sytuacja, w której rozważamy kawałek 1 na 1 i nie mamy lukru, który jest w stanie pokryć obszar o polu 1. Zauważmy jednak, że skoro wywołaliśmy się tak głęboko, to każde wywołanie było spowodowane faktem nieposiadania lukru o rozmiar większego. Oznacza to, że lukier się skończył, ale to by oznaczało, że pole lukru nie jest równe polu ciasta, co jest sprzeczne z założeniem.

7 Supermarket

Po pierwsze obróćmy całą przestrzeń o 45 stopni postrzegając (X, Y) jako $(X + Y, X - Y)$. Poprzez zaaplikowanie tego przekształcenia, zredukowaliśmy wybór kierunku do wyboru znaku w $(\pm D_i, \pm D_i)$. Dzięki temu problem można rozdzielić na dwa osobne podproblemy na osi OX oraz OY.

Czyli, sprowadziliśmy problem do przyporządkowania $+1$ lub -1 do s_i oraz s'_i, aby:

$$\sum_{i=1}^{N} s_i D_i = X + Y \quad \text{oraz} \quad \sum_{i=1}^{N} s'_i D_i = X - Y$$

Definiując $S = \sum_{i=1}^{N} D_i$, można dalej ułatwić nasze zadanie do znanego problemu plecaka (weźmy każdy element z $+1$ i wybierzmy elementy które będą odejmować dwukrotność ich wartości). Co można zapisać jako przyporządkowanie 0 lub $+1$ do t_i oraz t'_i, aby:

$$\sum_{i=1}^{N} t_i D_i = \frac{S + X + Y}{2} \quad \text{oraz} \quad \sum_{i=1}^{N} t'_i D_i = \frac{S + X - Y}{2}$$

Jeśli $(S + A + B)$ jest nieparzyste uzyskanie (X, Y) staje się niemożliwe do osiągnięcia.

Jak wcześniej wspomniano można zaaplikować klasyczną technikę plecaka, zapisując nasz problem jako:

$dp[x][y] = 1$ *jeśli można osiągnąć* $\sum_{i=1}^{x} t_i D_i = y$ *lub* 0 *wpp.*
$dp[x][y] = dp[x-1][y]$ *lub* $dp[x-1][y - D_i]$

Czas potrzebny do policzenia wszystkich stanów wynosi $O(NS)$ co dla danych tego zadania nie wystarcza. Aplikując strukturę danych *bitset* ze standardu C++ można przyspieszyć proces liczenia o 32 co wystarcza dla podanych limitów.

Zostaje odzyskanie odpowiedzi. Potrzebujemy do niej wszystkich wartości $dp[x][y]$, ale okazuje się że trzymanie ich w pamięci wystarcza na styk $2000 \times 1800 \times 2000 = 7.2 \times 10^9$ bitów $(= 900MB)$.

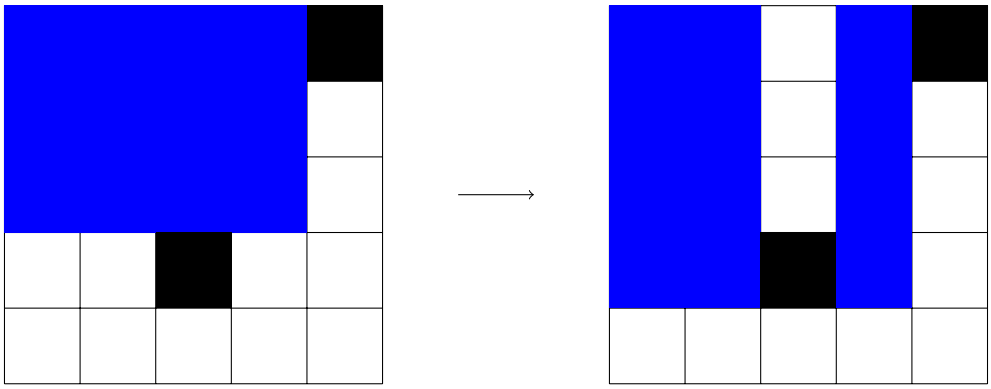
8 Wycinanie gofrów

Prostokąt nazwiemy nierozszerzalnym, jeśli nie da się go rozszerzyć w lewo, ani w prawo. Niech $ans[y][x]$ oznacza liczbę poprawnych wycięć prostokątów o wymiarach y na x, natomiast $ans2[y][x]$ liczbę poprawnych wycięć nierozszerzalnych prostokątów o tych samych wymiarach. Możemy wykorzystać $ans2$ do wyliczenia ans w czasie $O(NM)$ używając sum prefiksowych i następującego wzoru:

$$ans[y][x] = \sum_{i=x}^{M} ans2[y][i] * (i - x + 1)$$

Zatem sprowadziliśmy zadanie do wyliczenia wartości tablicy $ans2$. Będziemy iterować się po górnej współrzędnej prostokątów, którą nazwiemy $y1$. Dla każdej z tych współrzędnych zrobimy zamiatanie w

dół i niech aktualna współrzędna w zamiataniu nazywa się $y2$. Na miotle będziemy trzymać wszystkie nierozszerzalne prostokąty, których poziome boki są na wysokościach $y1$ oraz $y2$. Wydarzeniami w naszym zamiataniu będzie napotkanie '#', który sprawia, że danego nierozszerzalnego prostokąta z miotły nie można dalej przeciągnąć w dół. Należy wtedy zamienić ten prostokąt na mniejsze prostokąty (lub go usunąć jeśli miał szerokość 1).



Jeżeli nierozszerzalny prostokąt o szerokości x istniał na miotle na wysokościach $[i1, i2]$ to dla każdego i należącego do tego przedziału należy zwiększyć $ans2[i][x]$ o 1 co też można zrealizować sumami prefiksowymi. Każde z tych zamiatań można zrealizować w czasie $O(N + M)$, więc sumarycznie otrzymujemy czas $O(N(N + M))$.

9 Konkurs kulinarny

Zdefiniujmy

$dp[i][x]$ = minimalny koszt, aby prawidłowo ustawić pierwsze i dań oraz $l_i = x$.
$dp[i][x] = |l_i - x| + min_{x-(r_{i-1}-l_{i-1}) \leq x' \leq x+(r_i-l_i)} dp[i-1][x']$

Przeanalizujmy $dp[i]$ jako funkcję od x. Rysując ją na kartce można zauważyć, że jest wypukła oraz zbudowana z $2i + 3$ odcinków o nachyleniu $-i + 1, -i, \ldots, i, i + 1$ od lewej do prawej. Jest tak ponieważ, branie minimum na przedziale tylko rozszerza przedział gdzie nachylenie jest równe zero, a dodanie wartości bezwzględnej dodaje dwa punkty zmian nachylenia oraz zmniejsza na lewo i zwiększa na prawo nachylenie o 1.

Całe $dp[i]$ można przedstawić za pomocą $L_0, L_1, \ldots, L_i, R_0, R_1, \ldots, R_i$ oraz c:

- Na przedziale $(-\infty, L_i]$ z nachyleniem $-i - 1$
- Na przedziale $[L_i, L_{i-1}]$ z nachyleniem $-i$
- ...
- Na przedziale $[L_1, L_0]$ z nachyleniem -1
- Na przedziale $[L_0, R_0]$ z nachyleniem 0, na wysokości c
- Na przedziale $[R_0, R_1]$ z nachyleniem $+1$
- ...
- Na przedziale $[R_{i-1}, R_i]$ z nachyleniem i
- Na przedziale $[R_i, +\infty)$ z nachyleniem $i + 1$.

Obliczamy po kolei $dp[i]$ trzymając krzywą za pomocą dwóch kolejek zbiorów reprezentujących L_i oraz R_i. A na końcu wypisujemy c. Technika przedstawiania dp jako zbiór przedziałów z nachyleniem nazywa się *slope trick*, więcej o tym można znaleść tutaj: Link na USACO Guide.

```python
from heapq import heappush, heappop

n = int(input())

x = y = a = b = ans = 0
l, r = [], []

for _ in range(n):
    x, y = map(int, input().split())
    b += y - x

    heappush(l, -(y - a))
    heappush(r, y - b)

    if -l[0] + a > r[0] + b:
        t1 = -l[0] + a
        t2 = r[0] + b
        heappop(l)
        heappop(r)
        ans += t1 - t2
        heappush(l, -(t2 - a))
        heappush(r, t1 - b)

        a -= y - x

print(ans)
```